

CS-308-2014 Final Report

Automatic Driving Bot

Pulp Coding

TU-08

Vishnu Vardhan(100050066)

Priyank Chhipa(100050034)

Chandan Kumar(100050059)

Anirudh D(100050065)

Table of Contents

1. Introduction	3
2. Problem Statement	3
3. Requirements	3
3.1 Functional Requirements	3
3.2 Non-Functional Requirements	6
3.3 Hardware Requirements	6
3.4 Software Requirements	6
4. System Design	6
5. Working of the System and Test results	8
6. Discussion of System	10
7. Future Work	10
8. Conclusions	11
9. References	11

1. Introduction

Automated Driving Bot - A localization project which maps an unknown terrain. This project is intended to be used for various purposes in the greenhouse once the bot generates the map of the greenhouse. Thus, avoiding any human intervention.

First, the bot automatically creates a map of the entire greenhouse (by following black lines). Then a user can instruct it to move to a specific point in the greenhouse (by its coordinates). After receiving this instruction, the bot moves there using the shortest path possible. While moving, the bot also follows collision avoidance routine, to avoid collision with peers or any obstacle on the path.

The bot also keeps on monitoring the battery power levels. If it drops below a certain value, then it moves to the nearest battery charging dock to charge itself automatically.

2. Problem Statement

The aim of the project is to provide a decentralised driving (motion) mechanism for any kind of bot in any kind of arena. The protocols here makes each bot on its own, armed with a map which is stored in its memory. Even the collisions that arise during the motion of multiple bots in the arena are resolved locally between the two bots.

The project achieves the following goals to realised motion in an arena:

1. Mapping the Arena
2. Finding the shortest path to a location
3. Collision avoidance during motion
4. Battery Charge management

These protocols are implemented using FireBird V and details of each of these goals is given in subsequent sections.

3. Requirements

3.1 Functional Requirements

This is an automated driving bot. These are various functionalities to be supported by it:

1. **Map generation:** *Creates a map of the entire greenhouse (using blackline to follow). Each intersection is represented as a vertex in a graph.*

The paths in the Greenhouse are assumed to be in the form of a grid composed of blacklines. The bot starts at a corner (starting point), and does a Depth-First Search (DFS). Thus it visits all the vertices (intersections). After finding each new intersection, it adds it to the graph as a vertex. Thus, finally our map (graph) is ready. This graph has vertices in the form of (x,y) coordinates. The edges represent the presence of a path from one vertex to another.

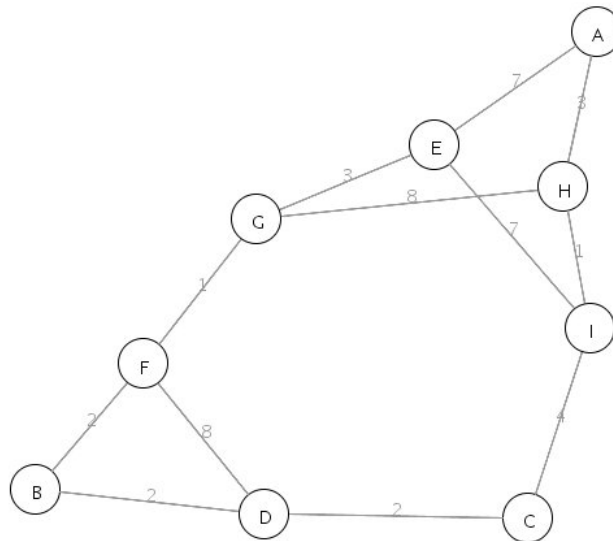


Figure: The whole grid is represented as a graph with edges as paths and vertices as intersections.

2. **Movement:** Moves to any instructed location using shortest path possible.

The bot receives an instruction (from a laptop) for moving to a point in terms of (x,y) coordinate. Then it finds the shortest path possible for the same using its current coordinate (x,y) and destination coordinates, after doing a BFS over the graph generated in the previous step. After this, it moves to the destination.

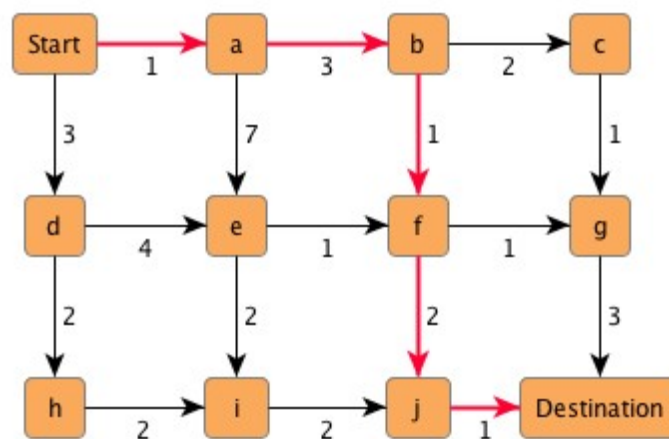


Figure: Applying BFS to find the shortest path to a destination

3. **Collision Avoidance:** Avoids collisions with other bots while doing the same.

The bot detects if there is an obstacle within a small range in its path using an IR-proximity sensor. If yes, it sleeps for a random amount of time. Then it marks the present edge (on which it is travelling right now) as blocked. Then it moves back to its previous location and finds another shortest path using BFS (as mentioned in the previous section). Thus, it moves to its destination. Sleeping for random amount of time also

ensures that there is no deadlock, in case there are multiple bots.

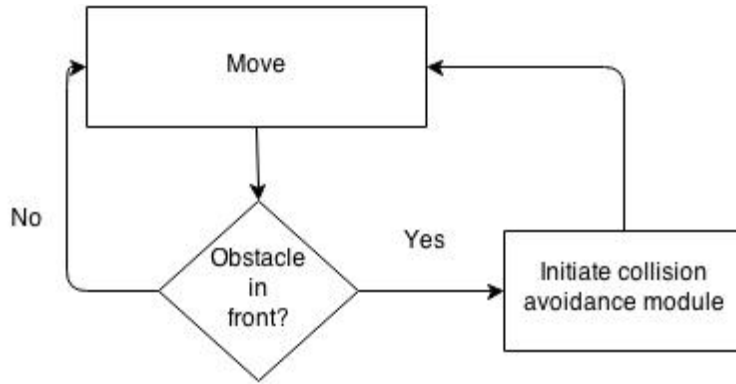


Figure: Collision detection module comes into play when the bot detects an obstacle in its front

4. **Automatic Charging:** Detects if battery is below a certain level. If yes, then moves to the charging dock before the battery discharges and charges automatically.

The bot regularly checks if the battery level is less than a pre-defined value (slightly more than the maximum battery power required to move from any point to the battery charging point- also a vertex in graph) using ADC. If yes, then it moves to the battery charging point (a pre-defined vertex in the graph), and charges itself.

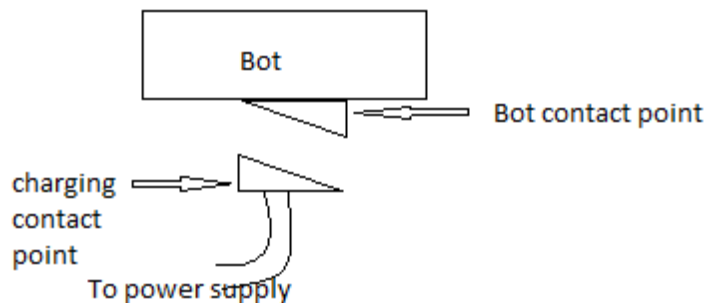


Figure: The charging of the bot explained (side view of plates)

3.2 Non-Functional Requirements

1. **Abstraction:** This implementation is essentially meant to enable the bot to travel to different locations (coordinates) in the map (grid). So, we can easily add other tasks that it has to perform at specific locations in the Greenhouse. For example, let's say we have a module *WaterPlant* which does

watering to the plants. By using our implementation for navigation and calling the module *WaterPlant* once we reach the required location, we can easily water all the plants at different locations (coordinates).

2. **Scalability and Reuse:** There are separate modules for different parts of functionality. This makes it convenient for reuse. For example, there are separate functions for map generation, navigation (using shortest paths), collision detection etc.
3. **Convenience:** The whole implementation can be easily installed on a FireBird V bot easily (as explained in the screencast).
4. **Distributed Architecture:** There is no centralized “control system” for the bots. This is the main advantage of this implementation. We can introduce (and remove) as many new bots as required. The functionality will not be affected as proper de-centralized collision avoidance mechanism is in place.

3.3 Hardware Requirements

1. FireBird V bot
2. XBee modules
3. Laptop (for debugging and testing if various modules are working properly, for example- map generation)

3.4 Software Requirements

1. Keil μ Vision IDE
2. X-CTU software
3. Python
4. C Compiler

4. System Design

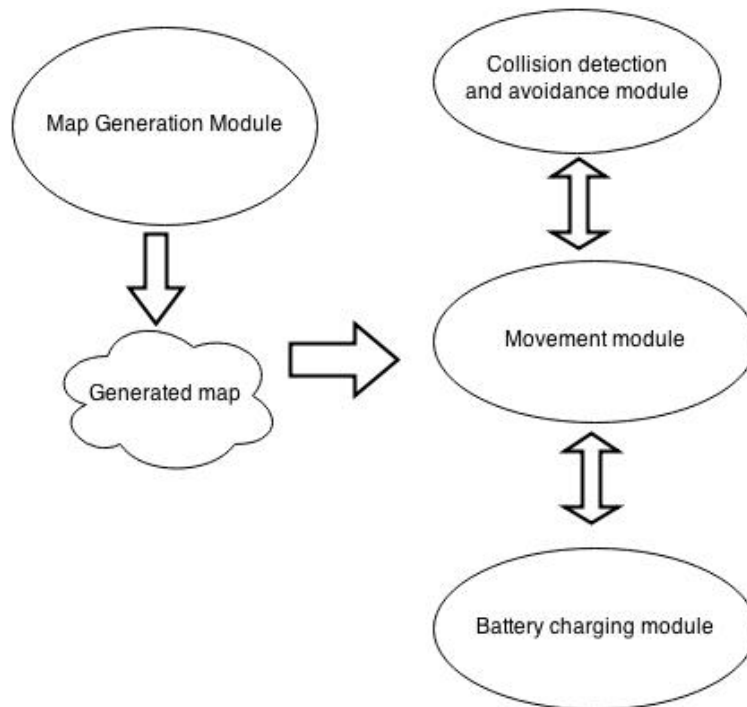


Figure: Interaction between different modules

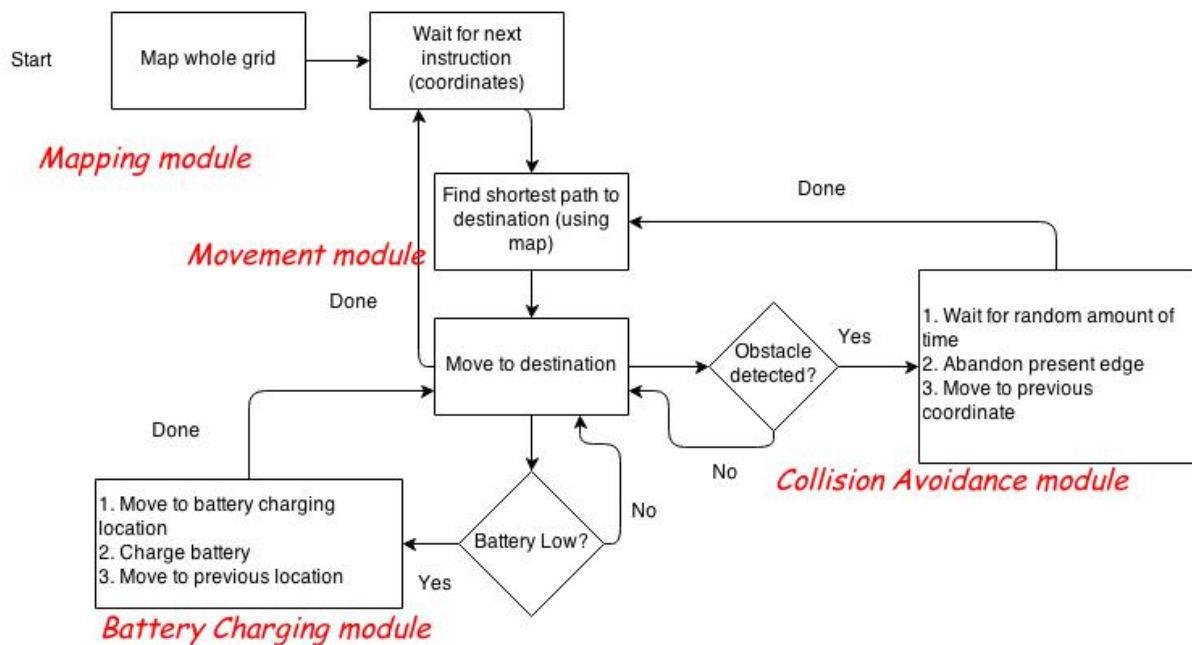


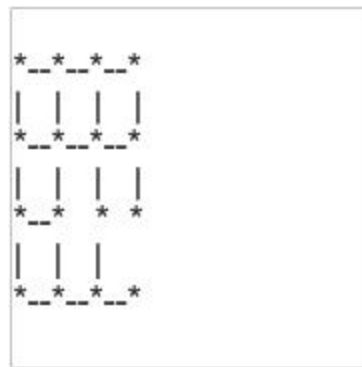
Figure: Overall design of the system

5. Working of the System and Test results

These are various modules (as described in Functionalities section) and their working:

1. Map generation:

The bot starts at the coordinate (0,0). Then it starts traversing all the intersections in the grid using DFS (Depth First Search - http://en.wikipedia.org/wiki/Depth-first_search) algorithm. The bot rotates at each intersection to find which edges are present. Using this approach, we have a 2-D array (graph) which contains the details of which edges are present and which are absent. Finally, the bot moves back to the (0,0) position, waiting for further instructions. A sample generated map looks like this (* is intersection, | and -- are edges):



2. Movement:

The bot receives the (x,y) coordinates one by one through the XBee module. Then it finds the shortest path to that location using BFS (Breadth First Search - http://en.wikipedia.org/wiki/Breadth-first_search) algorithm. Then it moves to the destination following this path.

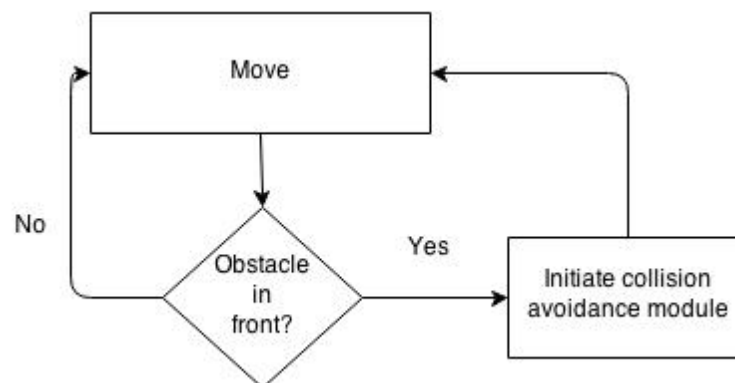


Figure: Collision avoidance module called whenever obstacle detected in front

3. Collision Avoidance:

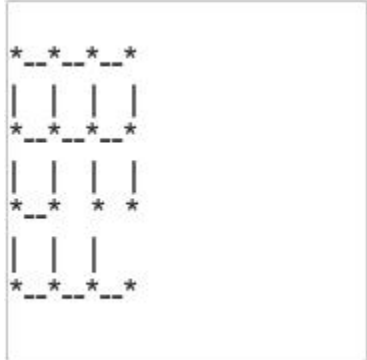
The bot keeps detecting for obstacles while navigating. Whenever it detects an obstacle, it sleeps for a random amount of time. Then it marks the present edge (on which it is travelling right now) as blocked. Then it

moves back to its previous location and finds another shortest path using BFS (as mentioned in the previous section). Thus, it moves to its destination. Sleeping for random amount of time also ensures that there is no deadlock, in case there are multiple bots.

4. Automatic Charging:

Throughout the whole process, the bot keeps monitoring its battery levels. Whenever it is less than a pre-defined value, the bot travels to the battery charging spot (using the shortest path) to charge itself. However, the battery charging dock could not be completed.

Testing Strategy

Functionality/module	Testing strategy
Map generation	<p>We ran our bot on a grid (composed of black lines) and displayed the graph (map) formed by our bot. These matched. The below was the generated map:</p>  <p>(* indicates intersections and and -- represent edges.)</p>
Movement	<p>We provided a series of input coordinates, after placing the bot at the starting point (after mapping was done). After each input, we checked if the bot has actually reached the provided coordinates. For example: (0,0), (1,2), (3,3) etc.</p>
Collision Avoidance	<p>We placed obstacles several times in front of the bot while moving. Each time the bot detected it, and found an alternative route, as per the algorithm.</p>
Automatic charging	<p>We reduced the battery level of our bot using various methods (movements, IR sensors, beeps and LED display). After the battery level reduced beyond our defined threshold for charging, the bot reached the charging location.</p>

6. Discussion of System

a) What all components of your project worked as per plan?

Map generation, Movement and Collision detection modules worked as planned.

Battery charging module could not be completed. The bot goes to the battery location if its battery drops below a certain level, but the battery charging dock is not complete.

b) What we added more than discussed in SRS?

We added visual display of the map generated by the bot. As soon as the map generation module is complete, the generated map can be viewed in a laptop.

We have also added a more robust collision avoidance mechanism than planned before.

In the SRS, the Greenhouse was assumed to be grid (complete). But later we have added te flexibility that any number of edges can be missing from the grid.

c) Changes made in plan from SRS:

Battery charging module could not be completed. The bot goes to the battery location if its battery drops below a certain level, but the battery charging dock is not complete.

We added visual display of the map generated by the bot. As soon as the map generation module is complete, the generated map can be viewed in a laptop.

We have also added a more robust collision avoidance mechanism than planned before.

Initially, the Greenhouse was assumed to be grid (complete). But later we have added te flexibility that any number of edges can be missing from the grid.

7. Future Work

The the current project is however generalistic (i.e. applicable to any bot) in nature is still limited to the sample arena chosen. So the first extention to the current project is to extend the path detection mechanisms along any size and kind.

While the current collision avoidance mechanism does serve the purpose of eventually reaching the destination; extended delays due to random sleeping is a concern and better algorithms could possibly be developed.

The logical extention to this project is to make the code, platform neutral i.e. independent of whether run in FireBird V or any other bot, the project should provide motion mechanisms and protocols smoothly.

8. Conclusions

This is a basic implementation for a FireBird V bot. And this can be used in combination with other projects, as this provides a nice mechanism for bot movements (using shortest paths and avoiding collisions with peers and still obstacles). For example, let's say we have a module WaterPlant which does watering to the plants. By

using our implementation for navigation and calling the module WaterPlant once we reach the required location, we can easily water all the plants at different locations (coordinates).

But there are many limitations in this implementation. We have made several assumptions like the grid is composed of square cells, the entire greenhouse is laid with black lines on white background etc. The present implementation can be extended to take care of these as well.

9. References

1. Depth-First Search- Wikipedia (http://en.wikipedia.org/wiki/Depth-first_search)
2. Breadth-First Search- Wikipedia (http://en.wikipedia.org/wiki/Breadth-first_search)
3. Pyserial Documentation (<http://pyserial.sourceforge.net/>)